The priority queue as an example of hardware/software codesign

Høeg, Flemming; Mellergaard, Niels; Staunstrup, Jørgen

Published in:
Proceedings of the Third International Workshop on Hardware/Software Codesign

Link to article, DOI:
10.1109/HSC.1994.336720

Publication date:
1994

Document Version
Publisher's PDF, also known as Version of record

The Priority Queue as an Example of Hardware/Software Codesign

Flemming Høeg, Niels Mellergaard, and Jørgen Staunstrup

Department of Computer Science
Technical University of Denmark
DK–2800 Lyngby, Denmark
e-mail: {fhoeg,nm,jst}@id.dtu.dk

Abstract

This paper identifies a number of issues that we believe are important for hardware/software codesign. The issues are illustrated by a small comprehensible example: a priority queue. Based on simulations of a real application, we suggest a combined hardware/software realization of the priority queue.

1 Introduction

A priority queue is a data structure with a simple interface which in many applications is a performance bottleneck. For example, in event driven simulators, the operations on a priority queue may account for a significant fraction of the computation time. Since the interface to a priority queue is simple and well defined, it seems like an obvious candidate for hardware realization while leaving other parts of the application in software. Despite its simplicity the priority queue illustrates several issues that are also relevant in more complex and less transparent examples of hardware/software codesign:

- the significance of an efficient interface between software and hardware components,
- the difference between optimal algorithms suited for software and for hardware realizations,
- the importance of estimating dynamic properties like communication traffic and execution profiles,
- the variety of aspects involved in hardware/software partitioning (speed, communication traffic, area, pin-count, etc.)

The paper uses the priority queue to explain and illustrate issues that we believe are of more general interest in codesign. In this paper the description of the specific priority queue example is merged with observations of more general codesign issues. The sections containing general observations are marked. The paper is organized as follows: first the functionality of the priority queue is specified (section 2), and a simple application is sketched (section 3). Section 4 describes efficient realizations of priority queues both in software and in hardware. Based on the properties of these realizations, section 5 discusses the issues involved in an efficient combination of software and hardware realization.

2 The priority queue

This section specifies the *functional* behavior of a priority queue. There are many ways to give such a specification. In connection with codesign, a main consideration is to specify in a way that is not biased towards either hardware or software.

Informally, a priority queue is a data structure that holds a set of elements from a domain with a total ordering relation, $<$, hence, any two elements can be compared.

Domains:

$$\begin{aligned} PQ &= ELEMENT\text{-}set \\ ELEMENT &= DATA \times PRIORITY \end{aligned}$$

Ordering:

$$\begin{aligned} &type\ <: ELEMENT \times ELEMENT \rightarrow BOOL \\ &type\ <: PRIORITY \times PRIORITY \rightarrow BOOL \\ &e_1 < e_2 \Leftrightarrow PRIORITY(e_1) < PRIORITY(e_2) \end{aligned}$$

There are three basic operations on the priority queue: *insert*, *remove*, and *minimum*. New elements may be inserted at any time (unless the queue is full). Similarly, the *smallest* of the elements currently in the queue may at any time be removed (unless the queue is empty). Hence, the design of a priority queue must ensure that when an element is removed, it is indeed

0-8186-6315-4/94 $04.00

the smallest (according to the ordering relation). Finally, the *minimum* operation returns the smallest of the elements currently in the queue without removing the element.

Operations:

```
type insert:  ELEMENT × PQ → PQ
type remove:  PQ → ELEMENT × PQ
type minimum: PQ → ELEMENT

insert(e, pq) ≡ pq ∪ {e}
remove(pq)    ≡ let m = minimum(pq)
                in (m, pq\{m})
∀e∈pq · minimum(pq) ≤ e ∧
        ∃e∈pq · minimum(pq) = e
```

At a given time, the *minimum* element is the element with the smallest priority value currently in the priority queue; this is the element extracted by the next remove operation.

The exact details of the interface to the priority queue are not specified; they are dependent on the environment in which the priority queue is going to be used, and on the realization (hardware/software) chosen. For a software realization, the priority queue could be specified as an abstract data type with the three operations implemented as procedure calls. On the other hand, for a hardware realization it could be more useful to define the interface in terms of communication ports, e.g., interface registers.

General observation. The specification illustrates an important issue, namely the importance of giving implementation independent specifications. Although we do not have a final proposal, the priority queue sketches an approach where the specification is given in two levels: the top level specifies functional properties in an implementation independent way (also ignoring issues like finite capacities of buffers, speed, etc.). The priority queue specification given above is an example of this. The bottom level gives one or more specifications of implementation dependent interfaces which may contain restrictions like capacity, response time etc. Both hardware and software implementations are discussed in section 4. Such a two-level approach is also found in VHDL where it is possible to define many different implementations of the same entity and in Larch [4] where it is possible to define many different programming language specific interfaces to the same abstractly specified component.

A wide span of different approaches to cospecification are currently investigated. Some use existing languages for hardware description, such as VHDL [2], or software, such as C [3], others advocate using an unbiased language [8], and there are also attempts to use graphical rather than symbolic specifications [5].

In the next section a specific application of a priority queue is presented. Possible realizations in both hardware and software are discussed for this application. It is illustrated that these hardware/software realizations are radically different. We believe that this is often the case. One reason is that software realizations are based on sequential algorithms, whereas hardware solutions often exploit parallelism.

3 An application

A priority queue can be used in an event driven simulator executing a model of some dynamic system (e.g., a network) where certain events (e.g., an activity in the network) lead to new events which must be simulated at a later time. The events awaiting simulation are stored in a priority queue where the priority is the starting time of the event. Hence, the smallest element in the priority queue is the next event to be simulated. This leads to the following main loop of the simulator:

```
while not empty do
  next:= remove;
  simulate(next)
```

The details of *simulate* are not discussed here, but it may lead to insertion of new events into the priority queue. A complete simulator contains many other parts, for example, a user interface, logging mechanisms etc., but from a performance point of view the loop shown above is a key part. Our experiments with the public domain simulator, Netsim, show that more than 30% of the run time is spent in the operations on the priority queue. The next section contains more details about the profiling.

General observation. In this application the goal is to get a speed-up for the cost of additional hardware. However, if the additional hardware is too expensive, e.g., several ASIC's are needed, or the speed-up is not significant, e.g., it is less than say 10 percent, then the conclusion may be that the codesign solution is not profitable. Such cost/benefit considerations are necessary to direct the development and evaluate the final solution. In different applications there might be other trade-offs considering, e.g., space or power consumption. Often it is not possible in advance to give a more precise formulation of the cost/benefit trade-offs than the informal remarks given above.

3.1 Profiling

This section presents the key observations obtained by profiling an existing simulator, Netsim, which is public domain and distributed by MIT and University of Washington, Seattle. Netsim was primarily chosen because the source code is available and its interface to the priority queue is very clean.

The following measurements are used later in this paper when discussing hardware/software partitioning and other codesign issues. A simulation example called nsfworld, distributed along with the Netsim simulator is modified (the example is modified by creating 8 parallel copies of the network) and used as input data to the simulator when profiling. When profiling the execution time, the simulator is used as it is distributed.

It must be realized that the profiling results lead to an optimization of the priority queue with respect to the environment in which it is used (nsfworld); it is therefore important that the simulation environment is representative of the practical environment of the priority queue. Ideally, a more careful investigation of the practical use of the priority queue should be carried out, however, to illustrate the codesign issues the single simulation example nsfworld is sufficient.

As it is illustrated in table 1 the operations on the

Operation	Number of operations	Total execution time
insert	2805132	10%
remove	2805132	20%
minimum	5301218	1%

Table 1: Priority queue operations in Netsim.

priority queue in the pure software solution (the Netsim simulator) take up a significant portion of the execution time. A major speed-up of the priority queue would therefore result in an increase in the overall performance of the simulator. In the rest of the profiling experiments a sequence of operations on the priority queue has been recorded and used as input for the profiling. A few specialized operations used to search-and-delete specific events have been filtered out.

The initial analysis showed that the maximum queue length during a simulation is 453, and the average queue length is 182. A more detailed analysis of the insertions into the priority queue is shown in table 2. It shows the distribution of priorities of new elements at the moment of insertion compared to the priorities of elements already in the queue, i.e., it shows *where* in the queue a new element belongs. The queue is divided into 10 segments of equal size (corresponding to the columns numbered 1–10, segment 1 is the first 1/10 of the queue, segment 10 is the last 1/10 of the queue). Line A shows the absolute distribution obtained by statically dividing the entire queue into segments; line B shows the relative distribution obtained by dynamically dividing the portion of the queue currently in use into segments. The segments are solely used for the profiling; they have no physical counterpart in the software and/or hardware realizations mentioned in this paper.

Seg.	1	2	3	4	5	6	7	8	9	10
A	72	11	12	3	1	1	1	0	0	0
B	56	10	7	2	2	3	2	4	11	3

A) Insertions relative to maximum queue length
B) Insertions relative to dynamic queue length

Table 2: Distribution of insertions into segments (%).

Table 2 shows that 95% of the elements are inserted into the first 30% (first three segments) of the queue. The reason for this could be that sometimes the queue is short, and in this situation, elements are always inserted into the first part of the queue. Another explanation could be that the event based simulator often schedules events to be fired soon compared to the other already scheduled events. This later possibility is analyzed by calculating relative to the dynamic queue length (B) instead of the maximum queue length (A). The table shows that most of the inserted elements (56%) are inserted into the first 10% of the queue even when calculating relative to the dynamic queue length (see line B). This also follows the distribution found in [10].

The observation that elements are usually inserted in the first segment calls for a fine grained analysis of the very first positions in the priority queue.

Position number	1	2	3	4	5
Insertion rate	43.6	7.0	0.6	0.1	0.1

Table 3: Insertions into the first five positions (%).

Table 3 gives a detailed overview of the insertion rate at the first five positions in the priority queue. Most of the scheduled events (43.6%) are sceduled as the first event.

The first few positions in the priority queue are very busy, and it is therefore obvious to seek a codesign solution that exploits this property. Another issue that might effect the realization is the dynamic variations inside the queue. For example, if the elements are going to be stored in a sorted array structure in hardware where only adjacent positions are connected, then it is important that the elements are not going to be moved back and forth over many positions. This internal activity is analyzed in the following measurements.

Table 4 shows the dynamic variations in queue length, when sampled for each k IO-operations (*insert* or *remove*). The figure shows that the dynamic

Sample interval (k)	Max. delta queue-size	Av. delta queue-size
10	10	0.91
100	100	2.75
1000	304	5.34
10000	310	11.22

Table 4: Variations in queue size.

variation in the queue length is very small. For example, when sampling 1000 IO-operations the average case is a change in queue size (delta queue-size) of only 5.34. This average case occurs, e.g., if there are 503 *insert* operations and 497 *remove* operations leading to a delta queue-size of 503–497=6, hence, the *insert/remove* ratio is almost 1/1.

The IO-operations are interleaved as shown in table 5. The table is calculated by ignoring *minimum*

Seq.length	1	2	3	4	5	6
Insert	67	33	<1	<1	<1	<1
Remove	69	23	6	1	<1	<1

Table 5: Successive *insert* and *remove* operations (%).

operations; the numbers show the percentage of the operations that occur in sequences of a given length, for example, 33% of the *insert* operations occur in sequences of length 2. It can be seen that the common case is one *insert* operation followed by one *remove* operation. 83% (=67+33/2) of the *insert* operations are immediately followed by a *remove*. These observations illustrate that the activity at the front of the queue is big, but small inside the queue.

General observation. This section presented some results from profiling an application of the priority queue. The results are utilized in section 5 to propose a hardware/software partitioning. There is a similar need for dynamic performance data in many other applications. In our case, a complete C program was available for doing the profiling. However, in other applications a complete implementation might not be available. It is therefore important to develop methods/tools for analyzing dynamic models early in the design phase.

4 Realization of a priority queue

Because of the importance of priority queues, many different realizations have been proposed both in software and in hardware. A few of these realizations are described below. Let n denote the maximum number of elements which can be contained in the queue at any one time.

4.1 Software realization

This section describes well known software realizations of a priority queue. A simple realization is obtained by maintaining a sorted list of the elements currently in the queue. Removing an element simply consists of copying and deleting the first element of the list. On the other hand, an insertion requires that the proper place of the new element is found in order to keep the list sorted; this requires $O(n)$ (worst case operation time).

There is no need to keep the elements of the priority queue completely sorted, all that is needed is to know the minimal element currently in the queue. The heap [9] is a suitable data structure for which it is

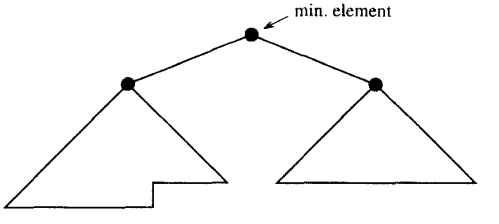

Figure 1: Heap structure.

possible to obtain operation times of $O(log(n))$ (worst case). The heap is a tree like structure (figure 1) where

care is taken to maintain a balanced tree. The Netsim simulator mentioned in section 3.1 uses a heap to represent the priority queue.

4.2 Hardware realization

This section describes a hardware realization of a priority queue. Such a realization provides faster operations on the queue because it is possible to do many steps of the computation in parallel. The interface to the priority queue consists of two registers: an input register, *in*, and an output register, *out* (figure 2). Each of these registers may contain an element from the domain of the priority queue, and each has additional status information indicating whether the register is full or empty. An element may be inserted in the queue when the input register is empty. Similarly, an element may be removed when the output register is not empty.

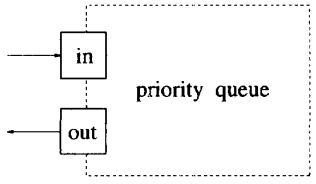

Figure 2: Hardware interface.

The hardware realization consists of an array of similar registers and a number of comparators which ensure that elements with a large priority value are moved backwards in the queue while small elements are kept in the front (figure 3). In fact, the smallest element currently in the queue is contained in the first (leftmost) register. It is therefore not necessary to wait for the priority queue to be completely sorted when extracting the smallest element. The detailed design is

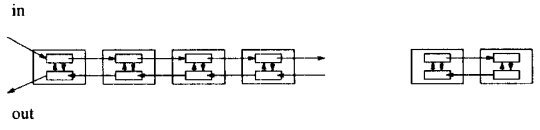

Figure 3: Hardware realization.

not given here, a systolic algorithm is presented in [6]. However, it is important to note that a hardware realization achieves constant operation times ($O(1)$ worst case). Hence, there is a potential for reducing the operation times significantly compared with a software realization.

General observation. The large difference between the hardware and the software realization points out the importance of computational models that do not favor one or the other. We believe that the sequencing found in most software makes it difficult to synthesize a good hardware solution from an optimized software description. There are alternative computational models avoiding explicit sequencing [1, 7].

5 Partitioning

This section discusses how to partition the event driven simulator into hardware and software components. To summarize the preceding sections, profiling of a large C program (Netsim) reveals that a few procedures implementing the priority queue operations account for more than 30% of the execution time, one could therefore hope for a performance improvement by placing some or all of the priority queue in hardware. However, a number of observations can be made from this simple example.

5.1 Hardware and software differences

An efficient software realization of a priority queue is a heap whereas the hardware realization is based on a linear array of registers with parallel comparison and movement of elements. It does not seem likely that a few simple transformations can synthesize one of these realizations from the other; they are based on *radically different algorithms*, in fact, they are based on *radically different computational models*. It seems that this might also be the case in many other applications of hardware/software codesign and that this should be accounted for in tools and techniques for partitioning and synthesis.

5.2 Integrated circuit constraints

The regular structure of the priority queue makes it well suited for realization as an integrated circuit. However, the area of a single stage of the queue is dominated by the size of the registers (one D-flip-flop ≈ 8 transistors) that is needed to store each element. A synchronous bit-serial realization is assumed; in case of a parallel realization the routing and the parallel comparators would occupy a major part of the total area. Each element consists of two 32-bit integers; one

32-bit integer for the priority value, and one for the event identifier (pointer to data). Note that one stage of the queue is capable of storing two data-elements. Therefore, approximately 1000 transistors are needed in each stage for the registers only, out of a total of approximately 1500. If it is possible to achieve 10000 transistors per mm^2 then 6 queue-stages per mm^2 is possible, or 300 stages per $50mm^2$. These figures indicate that it is possible (but expensive) to realize the whole priority queue with the proposed hardware solution (array of registers). In section 5.4 a better utilization of the hardware is suggested.

In the bit-serial realization, the power dissipation is higher than a parallel realization with similar transistor count and clock-rate, because in the bit-serial realization all elements are clocked/shifted forward all the time. All elements must be shifted through the bit-serial comparators to compute the internal state of the priority queue.

5.3 Hardware/software interface

To utilize the fast operation times achieved by a hardware realization the interface to the software must have a comparable speed. In case of the priority queue, small amounts of data is communicated frequently. The interface can be established by making the priority queue chip memory mapped. Inserting elements is, seen from the software side of the interface, similar to writing to a variable or memory location. For example, writing data (an element) to a specific address could be interpreted as *insert*, and reading a specific address could be interpreted as a *remove* operation. The latency of such an operation depends on the realization, but the latency is most likely greater than a normal memory operation.

Netsim simulations show that there are up to 10000 *insert* (and 10000 *remove*) operations per second (using a DecStation 5240, 40MHz MIPS). In this section an interface spending one micro second per transfer is examined (one transfer = 2x32bit). One micro second is arbitrarily chosen, but the idea is to examine how fast a bus that is needed. If one micro second is used per transfer, then 20000 micro seconds per second (2% of the CPU time) are spent waiting on the interface. During one micro second a CPU (like the MIPS) executes approximately 40 simple instructions. In general, if an application uses 80% of the CPU time on 20000 priority queue operations per second, and a reduction from 80% to 5% (total speed up = 4) is needed then 80000 operations per second are required (= 8% of the CPU time). The bus (interface) between a CPU and the main memory (RAM)

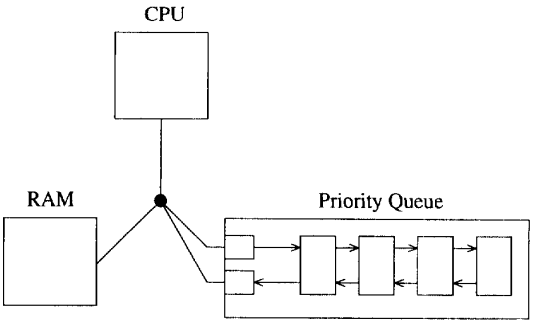

Figure 4: Hardware/software interface.

will in most cases be faster than one micro second and this bus will therefore be fast enough for the interface between the CPU (software) and the priority queue (hardware). However this example also shows that a latency greater than one micro second can not be tolerated, it will lead to poor performance. Note that the hardware is assumed to be fast, i.e., the latency involved in a transfer is dominated by the bus protocol.

General observation. The efficiency of the interface is very important. By choosing an inconvenient or slow interface one can easily loose the performance gain obtained by the hardware realization of the time crucial operations. We believe that a variety of different interfaces are needed to efficiently realize the large variety found in practical applications.

5.4 Size of a priority queue

It is very expensive to create a large priority queue using an integrated circuit realization like the one described in section 4.2 and 5.2. For example, a queue with a capacity of 1000 elements may require more than one chip. Furthermore, profiling indicates that this would also be an inefficient use of the hardware, since most of the changes to the queue take place towards the front, when used to schedule the events in Netsim.

Alternatively, the priority queue can be implemented by combining two priority queues in serial, with one fast priority queue (e.g., a hardware realization as an array of registers) at the front, and one slower but bigger priority queue with a better utilization of the area (e.g., a hardware realization implementing the heap algorithm) at the end. The idea

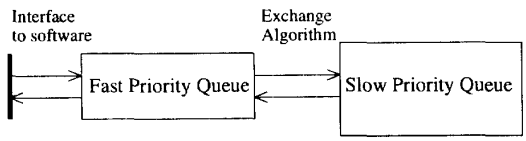

Figure 5: 2-stage priority queue.

is to utilize the fact that if an element with a small priority value is inserted into the first part of the priority queue, and the minimal element is removed, then the queue is still in order, without any access to the slow part of the queue. The second priority queue (slow but big) can be implemented using a standard RAM unit and a software like algorithm (for example a heap). Although both the algorithm/realization of the slow priority queue and the exchange algorithm between the two queues are important, possible solutions are not described further in this text. Tables 2 and 3 show that most of the changes in the priority queue take place towards the front, table 5 shows that most *insert* operations are followed by a *remove* operation (and visa versa). If the fast priority queue is able to buffer several elements then the exchange rate at the slow priority queue need only be, say, a tenth of the IO-rate at the software interface.

5.5 A hardware/software codesign

The partitioning of the problem/design into a hardware part and a software part can be made in many different ways, and the performance of the final design depends on this design decision. The bottleneck in such a design is probably the interface between the software and the hardware (i.e. the data-bus). A natural choice is to realize all priority queue operations in hardware. In the nsfworld simulation, see table 1, over 10 million transfers on the bus are then needed.

Size of software priority queue	Number of bus-transfers
0	10911482
1 (unbuffered)	5610264
1 (buffered)	2843272
3 (buffered)	2741690

Table 6: Bus-transfers with different sizes of the software priority queue.

Moving the first element (first priority queue-stage) into software the 5 million *minimum* operations are purely software function calls and no bus communication is needed for these operations. Note that this retains the constant operation times. If the first priority queue-stage (software-stage) is able to buffer elements (holding zero or one element) then more traffic on the bus can be saved, see table 6. The resulting 3-stage realization is illustrated in the following figure:

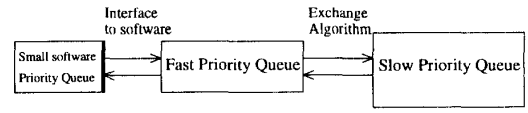

Figure 6: 3-stage priority queue.

The speed of the CPU and the hardware/software interface then determines the optimal number of stages to be realized in software.

Moving the first element into software is a small change in the design, but the number of bus transfers on the interface is reduced significantly. If the bus transfers are dominating the performance of the total system, then the performance of the realization changes accordingly.

General observation. The design shows that there is a variety of aspects involved in hardware/software partitioning (speed, communication traffic, area, pin-count, etc.). Therefore, a simple profiling of a software description is not sufficient. As illustrated above, it is necessary to estimate other dynamic properties than speed. It seems unlikely, that the partitioning of the priority queue could be generated automatically.

6 Conclusion

This paper identifies a number of issues that we believe are important for hardware/software codesign. The issues are illustrated by a small comprehensible example: a priority queue. Based on simulations of a real application, we suggest a combined hardware/software realization of the priority queue.

Acknowledgments

This work is partially supported by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Action

proj. no. 8135: "Cobra", and by the Danish Technical Research Council under the "Codesign" programme.

References

[1] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[2] W. Ecker. Using VHDL for HW/SW co-specification. In Gerry Musgrave, editor, *Proceedings of EURO-DAC '93, European Design Automation Conference*, pages 500–505. IEEE Computer Society Press, September 1993.

[3] Rajesh K. Gupta and Giovanni De Micheli. System level synthesis using re-programmable components. In *The European Conference on Design Automation*, pages 2–7. IEEE, March 1992.

[4] John V. Guttag, James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[5] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] Charles E. Leiserson. *Area-Efficient VLSI Computation.* ACM Doctorial Dissertation Awards. The MIT Press, 1983.

[7] Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[8] Jørgen Staunstrup. Towards a common model of software and hardware components. In Jerzy Rozenblit and Klaus Buchenrieder, editors, *Codesign: Computer Aided Software/Hardware Engineering.* IEEE Press, 1994.

[9] J.W.J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[10] F. Paul Wyman. Improved event-scanning mechanisms for discrete event simulation. *Communications of the ACM*, 18(6):350–353, June 1975.